AD-A062 533 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
SPECIFICATION AND PROOF TECHNIQUES FOR SERIALIZERS.(U)
AUG 77 R ATKINSON, C HEWITT N00014-75-C-0522

UNCLASSIFIED AI-M-438 NL

1 OF 1
AD
A062533

END
DATE
FILMED
3-79
DDC

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER AIM 438	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Specification and proof techniques for serializers		5. TYPE OF REPORT & PERIOD COVERED memorandum 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Russell Atkinson ~~and~~ Carl Hewitt		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0522 15 N00014-75-C-0522
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE 11 August 1977 13. NUMBER OF PAGES 35
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

12 39 p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

14 AI-M-438

18. SUPPLEMENTARY NOTES

None 9 Technical memo

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

specification language
serializer
synchronization
computation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents an implementation mechanism, specification language, and proof techniques for problems involving the arbitration of concurrent requests to shared resources. This mechanism is the serializer, which may be described as a kind of protection mechanism, in that it prevents improper orders of access to a protected resource. Serializers are a generalization and improvement of the monitor mechanism of Brinch-Hansen and Hoare.

Serializers attempt to systematize and abstract desirable structural features

20. of synchronization control structure into a coherent language construct. They represent an improvment in the modularity of synchronization over monitors in several respects. Monitors synchronize requests by providing a pair of operations for each request type (examples are STARTREAD/ENDREAD and STARTWRITE/ENDWRITE for the readers-writers problems). Such a pair of operations must be used in a certain order for the synchronization to work properly, yet nothing in the monitor construct enforces this use. Serializers incorporate this structural aspect of synchronization in a unified mechanism to guarantee proper check-in and check-out. in scheduling access to a protected resource, it is often necessary to wait in a queue for a certain condition before it continues execution. Monitors require that a process waiting in a queue will remain dormant forever, unless another process explicitly signals to the dormant process that it should continue. Serializers improve the modularity of synchronication by providing that the condition for resuming execution must be explicitly stated when a process enters a queue, making it unnecessary for processes to signal other processes. Each process determines for itself the conditions required for its further execution.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 438

August 1977

SPECIFICATION AND PROOF TECHNIQUES FOR SERIALIZERS

Russell Atkinson
Carl Hewitt

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0522.

SECTION I --- ABSTRACT

This paper presents an implementation mechanism, specification language, and proof techniques for problems involving the arbitration of concurrent requests to shared resources. This mechanism is the serializer, which may be described as a kind of protection mechanism, in that it prevents improper orders of access to a protected resource. Serializers are a generalization and improvement of the monitor mechanism of Brinch-Hansen and Hoare.

Serializers attempt to systematize and abstract desirable structural features of synchronization control structure into a coherent language construct. They represent an improvement in the modularity of synchronization over monitors in several respects. Monitors synchronize requests by providing a pair of operations for each request type [examples are STARTREAD/ENDREAD and STARTWRITE/ENDWRITE for the readers-writers problems]. Such a pair of operations must be used in a certain order for the synchronization to work properly, yet nothing in the monitor construct enforces this use. Serializers incorporate this structural aspect of synchronization in a unified mechanism to guarantee proper check-in and check-out. In scheduling access to a protected resource, it is often necessary to wait in a queue for a certain condition before it continues execution. Monitors require that a process waiting in a queue will remain dormant forever, unless another process explicitly signals to the dormant process that it should continue. Serializers improve the modularity of synchronization by providing that the condition for resuming execution must be explicitly stated when a process enters a queue, making it it unnecessary for processes to signal other processes. Each process determines for itself the conditions required for its further execution.

The behavior of a serializer is defined axiomatically in terms of causal and incidental relations among events using the actor message-passing model of computation. Different versions of the "readers-writers" problems are used to illustrate how the structure of a serializer corresponds in a natural way to the structure of the specification of synchronization problems.

In this paper we present specification and proof techniques using partial orders on computational events for dealing with problems involving fairness, starvation, and guaranteed concurrency. Our techniques represent a significant advance over previously developed techniques using global states.

SECTION II --- PARALLELISM in PROBLEM SOLVING

Serializers have been developed as a modular arbitration primitive for actor systems to aid investigating the issues of parallelism in problem solving. Conceptually an actor is an object that has both procedural and data aspects. The behavior of a primitive actor such as a serializer is defined by the ordering relationships among the events caused by the actor.

We now feel that parallelism may have a more important role than previously realized in explicating the structure of higher level symbolic processing. Until recently it has been widely accepted that parallelism is not suited for the higher level symbolic processing of problem solving. Most psychological evidence seems to point to individual humans as being almost entirely serial in their high level problem solving.

Recently the development of actor message passing semantics has brought about a shift in our paradigm for problem solving. Early programs written to be expert in some domain were thought to be analogous to an individual human expert who was expert in the domain. Most programs were developed on the basis that there should be a single unified coherent intelligence directing all aspects of the problem solving in a serial fashion. The development of the actor model of computation has encouraged us to develop a paradigm based on _a society of experts communicating by passing messages_. This switch in paradigm has provided us with a rich source of ideas for problem solving strategies. We are attempting to develop a _dialogue style_ of programming which places its emphasis on the modular distribution of knowledge and clean means of communication between pieces of knowledge. Thinking and programming in this new paradigm has in turn caused us to re-evaluate the case for parallelism. We note that societies often make good use of parallelism for a variety of purposes.

The additional programming burden imposed by parallelism is the task of arbitrating the activities of modules running in parallel. By analyzing the structure of problem solvers that attempt to use parallelism at the highest levels of problem solving, we hope to further explicate the structure of problem solving. The explication of a modular arbitration primitive in this paper should contribute toward that aim.

We see a need for the development of language constructs that are at least partially chosen for their provability. A language feature providing synchronization should be designed to provide usable axioms about the possible orders of events in a program. The language feature should guarantee that conditions needed to prove properties of programs are explicit in the axioms for the language feature.

Serializers have been designed to facilitate the proof that schedulers implemented using them satisfy their specifications. The specifications of a protected resource typically involve stating both integrity and scheduling constraints. An integrity specification typically takes the form of a consistency constraint. A typical example of an integrity specification might be that the position and velocity of an airplane must be recorded for the same instant of time. A scheduling specification typically takes the form of a constraint on the time order of certain events. A typical example of a scheduling specification is that if two requests to write in a data baseare received in a certain order then the first request received will be honored before the other. We would like to be able to demonstrate how implementing protected resources using serializers makes it easier to prove that they satisfy their specifications. In particular, we would like to develop techniques for proving that schedulers implemented using serializers guarantee a reply to each request received. Guaranteeing that a reply will be sent for every request received is a stronger and more useful property than merely being free of deadly embrace, which is the scheduling specification most extensively treated in the literature on synchronization.

SECTION III --- SERIALIZERS

III.1 --- Concept of a Serializer

In this section we will describe an abstract mechanism called a serializer for guaranteeing the integrity of a protected resource. The mechanism is an abstraction and encapsulation of the method commonly used in operating systems. A detailed analysis of the facilities needed will be used to motivate our design decisions.

A serializer bears an analogy to the front desk of a hospital in that only one person can check in or out at a time. The front desk of a hospital serves to schedule the entrance and exit of people in the hospital. Entering or leaving the hospital is impossible without checking through the front desk. Various queues are maintained for people who are waiting. In addition records are maintained of where people are within the hospital.

Serializers are modular in the sense that they can be constructed to encase the resource to be protected in such a way that it can only be accessed by passing through the serializer. A serializer should be constructed to surround the protected resource in such a way that it is impossible to accidently avoid passing through it when using the protected resource. We shall avoid in this paper the issues involved with exactly how one guarantees that a serializer has sole possession of a resource, or even if cooperating serializers might share access to a resource. The reader may assume that every serializer we deal with in this paper has sole access to the encased protected resource.

We can diagram how a serializer can be used to schedule access to a protected resource P as follows:

```
SERIALIZER-REQUEST
        |
+-------|---------------------------------------+
|       V                                       |
|                                               |
|    ENQUEUE                                    |
|       |                                       |
|  +----|----+             JOIN-CROWD           |
|  |    V    |                  |               |
|  |         |          +-------|-------+       |
|  | DEQUEUE |          |       V       |       |
|  |    |    |          |               |       |
|  +----|----+          | +-----------+ |       |
|       |               | |           | |       |
|       V               | | RESOURCE  | |       |
|                       | |           | |       |
|                       | +-----------+ |       |
|                       |               |       |
|                       |  LEAVE-CROWD  |       |
|                       |       |       |       |
|                       +-------|-------+       |
|      EXIT                     |               |
|       |                       V               |
|       |                                       |
+-------|---------------------------------------+
        |
        V
```

Diagram of Serializer Encasing a Protected Resource

Each arrow in the above diagram is labeled with the kind of computational event it represents. The events fall into two disjoint categories that are totally ordered in time: those which **GAIN-POSSESSION** and those which **RELEASE-POSSESSION** of the serializer. Each event in the former category subsequently causes an event in the latter category to occur. Furthermore after a **GAIN-POSSESSION** event has occurred, then another such event will not occur until after the former has caused a **RELEASE-POSSESSION** event. A typical sequence of events occurring in the use of the protected resource P begins with a **SERIALIZER-REQUEST** event in which the serializer receives a message M which is intended for the protected resource P. The request must eventually result in an **ENTRY** event which gains possession of the serializer. A **GUARANTEE** request can be used in order to wait until some condition is true before proceeding. Such a request releases possession of the serializer. If a reply is received for the **GUARANTEE** request then it will be called an **ESTABLISHED** event because the condition is guaranteed to have been established at the time of the reply. Thus each **ESTABLISHED** event regains possession of the serializer at a point in time when the condition is guaranteed to be true. The above sequence of **GUARANTEE** and **ESTABLISHED** events may occur a number of times in order to sequentially guarantee a number of conditions in succession. When the proper condition for using a protected resource has been established then possession of the serializer can be released by a **JOIN-CROWD** event which records that there is another process in the crowd using P. Next the message M is delivered to the protected resource P in a **RESOURCE-REQUEST** event. Eventually the protected resource P may produce a reply R to the request which will be called a **RESOURCE-REPLY** event. The **RESOURCE-REPLY** will eventually result in a **LEAVE-CROWD** event which regains possession of the serializer and records that the process is no longer in the crowd using P. The next event is an **EXIT** event, which releases possession and causes a **SERIALIZER-REPLY** event in which the message R is sent as the reply to the original **SERIALIZER-REQUEST** event.

Serializers derive their name from the fact that all of the events that gain and release possession of the serializer are totally ordered (serial) in time. We assume that every serializer is written such that an event gaining possession is always followed by one releasing possession (usually this is trivial to demonstrate). In the above diagram the interior of the serializer has two "holes", in which a process may temporarily release possession of the serializer. The purpose of the hole entered by **ENQUEUE** is to wait for some condition to become true. The purpose of the hole entered by **JOIN-CROWD** is to allow parallelism in the use of protected resources by releasing the serializer to another process. There may be any number of holes of either variety.

To understand the behavior of serializers, one must understand the ways that possession of a serializer is gained and released. There are three ways to gain possession of a serializer:

> An **ENTRY** event, which gains possession as a result of a **SERIALIZER-REQUEST** event.
>
> An **ESTABLISHED** event, which regains possession as a result of a **GUARANTEE** request with a condition established to be true.
>
> A **LEAVE-CROWD** event, which regains possession as a result **RESOURCE-REPLY** event from a protected resource.

There are three ways to release possession of a serializer:

> A **GUARANTEE** event, which occurs in order to guarantee that some condition is true before continuing execution.
>
> A **JOIN-CROWD** event, which records that a process is using the protected resource.
>
> An **EXIT** event, which causes a reply to the original **SERIALIZER-REQUEST** event.

For any given serializer and process after an **ENTRY** event and before the corresponding **EXIT** event, exactly one of the following two conditions will hold:

> The process is in sole possession of the serializer (executing in the shaded region of the diagram).
>
> The process has released possession in order to wait for some condition before proceeding or to join a crowd of processes executing in some protected resource.

SECTION IV --- SERIALIZER CONSTRUCTS

In this section we present the language constructs used in the serializer mechanism. They have been developed to facilitate the implementation of the abilities enumerated above. We should note that while a LISP-like syntax is used, we regard the choice of syntax as minor.

IV.1 --- Creation

A serializer is constructed by an expression of the form

```
(create_serializer
   (queues: collection_of_queues_for_the_serializer)
   (crowds: collection_of_crowds_for_the_serializer)
   (entry: body_of_serializer))
```

The queues are used to provide first-in-first-out service to processes waiting for some condition in order to continue execution. The crowds are used to record which protected resources are in use.

If an actor constructed by an expression of the form given above is sent a message M in a **SERIALIZER-REQUEST** event then M will eventually be sent to body_of_serializer in an **ENTRY** event which gains possession of the serializer. At most one process can be in possession of a serializer at one *time.* The queues and crowds for the serializer relate to its internal working and are explained in greater detail below.

IV.2 --- Queues

Queues are provided to allow a process to wait until some condition is met before proceeding further. Serializers provide an *enqueue* command which has the following syntax to meet this need:

```
(enqueue the_wait_queue
      (guarantee: the_condition)
      (then: the_continuation))
```

A process executing the above command immediately releases possession of the serializer with a **ENQUEUE** the_wait_queue to **GUARANTEE** the_condition event. It does not regain possession and continue with execution of the_continuation with an **ESTABLISHED** the_condition event until all of the following pre-requisites hold:

1: All the previous **GUARANTEE** requests in the_wait_queue have received replies that the condition was **ESTABLISHED**; i.e. the process is at the front of the queue.

2: the_condition is true.

3: A **JOIN-CROWD**, **EXIT**, or **GUARANTEE** event has just occurred releasing possession of the serializer.

Note that all of these pre-requisites must be simultaneously satisfied before execution will continue with the_continuation.

It may be the case that there is more than one process which satisfies the above pre-requisites because each is at the front of a different queue. In this case it is not obvious which process should gain possession next. We recommend that serializers be constructed so that they satisfy the following property:

4: There is no other process such that the above three conditions hold.

If all four conditions hold then for a process, the it is guaranteed to get possession of the serializer next before any further **ENTRY** or **LEAVE-CROWD** events occur. In other words, conceptually at least, the condition that must be guaranteed for further execution is evaluated for each process that is at the front of a queue every time possession of the serializer is released. If there is only one whose condition is true then it gets possession next.

The condition in the *guarantee:* clause of the *enqueue* command is potentially any Boolean expression without side effects. The expression is evaluated whenever possession of the serializer is yielded. We have found one particular form of expression to be quite useful, which is a test for emptiness of queues or crowds. This is written as:

(*empty:* queue$_1$ queue$_2$ ...)

Each queue listed must be empty for the expression to be true. The evaluation of an expression of the above form has no side-effects. It simply calculates the Boolean value for the expression.

We wish the_condition to be guaranteed to be true when execution continues with the_continuation. This depends on several constraints:

1: The evaluation of the_condition is free of side-effects.

2: The value of the_condition must be unaffected by any execution by any process which does not possess the serializer.

Given these constraints, we can say that the_condition must be true when execution resumes with the_continuation of an *enqueue* command.

IV.3 --- Relaying Messages

Within a serializer it is necessary to be able to temporarily release possession of the serializer in order to relay the message to the protected resource and then later regain possession with the reply from the protected resource. A command of the following form accomplishes this by transmitting a_message to a_protected_resource:

```
(relay_to a_protected_resource
   (message: a_message)
   (thru: a_crowd)
   (then_to: continuation_for_reply))
```

An entry is made in a_crowd to record the presence of a process in a_protected_resource and then possession is yielded. After a_protected_resource has replied to a_message and possession of the serializer has been regained by a LEAVE-CROWD event then the entry is removed from a_crowd and the reply received is sent to continuation_for_reply. We have observed that in many applications the reply received from a_protected_resource is immediately returned as the value of the serializer in an EXIT event. Therefore, we adopt the convention that if the *then_to:* clause is omitted from a *relay_to* expression, then an EXIT will be performed with the message received from the protected resource.

During the time between the JOIN-CROWD and the LEAVE-CROWD events, the entry is a member of a_crowd. Thus by inspecting the various crowds of a serializer it is possible to determine which resources currently have processes executing within them. Crowds provide a better abstraction than integer counts for keeping records of which processes are currently executing in protected resources.

SECTION V --- BEHAVIORAL PROPERTIES OF SERIALIZERS

The properties of serializers are stated somewhat informally in this paper since we believe that serializers aid intuitive reasoning about parallelism. A more rigorous treatment is possible, but is beyond the scope of this paper.

Behavioral properties of serializers can be stated in terms of events and relations between events. We shall use the notation

$$E_1 \dashrightarrow E_2$$

to indicate that the event E_1 precedes the event E_2. The precedes relationship is an acyclic partial order. The events of processes that do not interact are not ordered.

In the rest of the paper we will require that the protected resource is well-behaved in the sense that for each request sent to the resource exactly one reply will be received.

Another requirement we will make is that every process that comes into possession of a serializer will eventually release possession. The intent is to exclude behaviors where the serializer is locked up forever by a process which is performing an infinite computation while in possession. We believe that this condition will usually be trivial to satisfy in practice since the code in a serializer simply has to decide whether to wait for some condition or join some crowd of processes executing in a protected resource. This decision must be made as efficiently as possible in order to maximize the thruput of the serializer. Otherwise the serializer can seriously degrade the efficiency of a system by becoming a bottleneck.

V.1 --- Property of Mutual Exclusion

The most fundamental property of a serializer is that processes mutually exclude one another from possession so that at most one process has possession at any given instant. For any given serializer there is a total ordering in time for all of the ENTRY, GUARANTEE, ESTABLISHED, JOIN-CROWD, LEAVE-CROWD, and EXIT events of that serializer. A process gains possession of a serializer starting with an ENTRY, ESTABLISHED, or LEAVE-CROWD event, and retains possession until it releases possession with a GUARANTEE, JOIN-CROWD, or EXIT event. We will use subscripts to indicate distinct invocations of a serializer. The property of mutual exclusion of possession of the serializer can be stated in terms of events as follows:

Either the i-th possession precedes the j-th possession

$$\begin{tikzcd}
\text{GAIN-POSSESSION}_i \arrow[d] & \\
\text{RELEASE-POSSESSION}_i \arrow[r, dashed] & \text{GAIN-POSSESSION}_j \arrow[d] \\
 & \text{RELEASE-POSSESSION}_j
\end{tikzcd}$$

or the j-th possession precedes the i-th.

$$\begin{tikzcd}
 & \text{GAIN-POSSESSION}_j \arrow[d] \\
\text{GAIN-POSSESSION}_i \arrow[d] & \text{RELEASE-POSSESSION}_j \arrow[l, dashed] \\
\text{RELEASE-POSSESSION}_i &
\end{tikzcd}$$

where a GAIN-POSSESSION_i event is either an **ENTRY_i**, **ESTABLISHED_i**, or **LEAVE-CROWD_i** event; and $\text{RELEASE-POSSESSION}_i$ is the next event after **GAIN-POSSESSION_i** which is a GUARANTEE_i, JOIN-CROWD_i, or **EXIT_i** event.

V.2 --- Gaining Possession

We would like to guarantee that any process that sends a request or reply to a serializer must eventually gain possession of the serializer. This property is satisfied by any serializer that has no loops in which possession of the serializer is not released in the loop. It is not clear that it is ever useful to violate this restriction. All of the serializer examples in this paper trivially satisfy this restriction because they have no loops at all.

If the above restriction is satisfied then any **SERIALIZER-REQUEST** or **RESOURCE-REPLY** event must eventually result in a GAIN-POSSESSION event. More precisely, if there is a SERIALIZER-REQUEST in the history of a computation then it is followed by an ENTRY event. Furthermore if there is a RESOURCE-REPLY event in the history of a computation then it is followed by a LEAVE-CROWD event.

V.2.a --- First Come First Served for Entry

Since serializers are designed to implement scheduling of access to protected resources it must be possible for them to observe the order of arrival of requests to the serializers in order to carry out certain scheduling tasks. Thus we provide that requests for entry into the serializer will be served in the order in which they arrive at the serializer. In terms of events this can be formalized by supposing that $\textbf{SERIALIZER-REQUEST}_i$ and $\textbf{SERIALIZER-REQUEST}_j$ are two events such that such that the first arrives before the second:

$$\textbf{SERIALIZER-REQUEST}_i \text{ --> } \textbf{SERIALIZER-REQUEST}_j$$

The next event after $\textbf{SERIALIZER-REQUEST}_i$ is $\textbf{ENTRY}_i$ and the next event after $\textbf{SERIALIZER-REQUEST}_j$ is $\textbf{ENTRY}_j$. We require that these two events be related as follows:

$$\textbf{ENTRY}_i \text{ --> } \textbf{ENTRY}_j$$

V.2.b --- First Come First Served for Re-Entry

Similarly it must be possible for a serializer to observe the order of arrival of replies to requests sent to protected resources. Thus if $\textbf{RESOURCE-REPLY}_i$ precedes $\textbf{RESOURCE-REPLY}_j$ so that

$$\textbf{RESOURCE-REPLY}_i \text{ --> } \textbf{RESOURCE-REPLY}_j$$

then we require that

$$\textbf{LEAVE-CROWD}_i \text{ --> } \textbf{LEAVE-CROWD}_j$$

V.3 --- Properties of Guaranteed Conditions

V.3.a --- The Guaranteed Condition is True if Execution Continues

Let C be the condition guaranteed in an event of the form

$$\textbf{ENQUEUE}_q\textbf{-GUARANTEE}_C$$

which is caused by executing an expression of the form

```
(enqueue q
   (guarantee: C)
   (then: ...))
```

If execution of the process continues, then the next event of the process is of the form $\textbf{ESTABLISHED}_C$ and C is true at the instant of this event.

V.3.b --- Internal Queues are First In First Out

Suppose that q is an internal queue of the serializer S and that there are two events such that

$$\mathrm{ENQUEUE}_q\text{-}\mathrm{GUARANTEE}_{C1} \dashrightarrow \mathrm{ENQUEUE}_q\text{-}\mathrm{GUARANTEE}_{C2}$$

[i.e. such that the former precedes the latter in the total ordering of the serializer] and that $\mathrm{ESTABLISHED}_{C2}$ is the next event after $\mathrm{ENQUEUE}_q\text{-}\mathrm{GUARANTEE}_{C2}$. Then there is an event $\mathrm{ESTABLISHED}_{C1}$ which is the next event after $\mathrm{ENQUEUE}_q\text{-}\mathrm{GUARANTEE}_{C1}$ such that

$$\mathrm{ESTABLISHED}_{C1} \dashrightarrow \mathrm{ESTABLISHED}_{C2}$$

which says that $\mathrm{ENQUEUE}_q\text{-}\mathrm{GUARANTEE}_{C1}$ was served before $\mathrm{ENQUEUE}_q\text{-}\mathrm{GUARANTEE}_{C2}$ since both guarantee requests were placed in the same queue.

V.3.c --- Priority for Waiting Processes

Each time possession of a serializer is released, waiting processes are given the opportunity to continue execution. This property of serializers simplifies proofs that a scheduler guarantees replies to requests received and increases the responsiveness of schedulers by allowing waiting processes to proceed as soon as possible. Roughly speaking, if there are any waiting processes "ready to go" when possession of a serializer is released then the next event to gain possession of the serializer must be an ESTABLISHED event which gives one of those processes possession. In terms of events, a process will be said to be "ready to go" at the instant of a RELEASE-POSSESSION event if it is waiting because of a previous $\mathrm{ENQUEUE}_Q\text{-}\mathrm{GUARANTEE}_C$ event, but the corresponding ESTABLISHED event has not yet occurred and the following properties hold:

1: The condition C is true.

2: The process is at the front of the queue. Therefore all previous events that waited for some condition on Q have already continued with their condition ESTABLISHED.

The above properties give internal queues priority over external queues.

SECTION VI --- RELATIONSHIP TO SEMAPHORES

VI.1 --- Mutual Exclusion

One of the most common uses of semaphores is to implement mutual exclusion of execution in protected resources. It is relatively easy to implement mutual exclusion using a semaphore. The idea is for each process to perform a *P* operation on the semaphore before using the resource and then to perform a *V* operation when finished using the resource. The program mutual_exclusion_1 given below can be used to construct systems that insure that a resource does not receive any messages while still processing a previous message. Thus processes are guaranteed to mutually exclude each other from overlapping execution in the protected resource. This simple example is presented to illustrate more concretely the concept of encasing a resource in a serializer.

```
(mutual_exclusion_1 =resource) ≡                    ;mutual exclusion of a resource is enforced by
   (let (mutex = (create_binary_semaphore))        ;constructing a new binary semaphore called mutex
    in (≡> =a_message                    ;then returning an actor such that whenever it receives a message
         (P mutex)                                           ;performs a P operation on mutex
         (let (result = (resource <= a_message))         ;then sends the message to the resource
          in                                                ;such that after the result is received
            (V mutex)                                   ;a V operation is performed on the semaphore
            result)))                                          ;and the result is returned
```

Semaphores are a very primitive synchronization method which can be used to implement the facilities needed by modular schedulers. In some ways semaphores are analogous to the goto construct which can be used to implement the control structures needed for modular programming. Serializers abstract the control structure of schedulers such as the simple one presented above. They can be used to increase the modularity of implementations by making the structure of the implementation more closely match the structure of the task to be accomplished. In this way the synthesis of schedulers from specifications is facilitated because serializers provide facilities for directly implementing common aspects of specifications for schedulers. Furthermore proofs that implementations satisfy their specifications is facilitated because the structure of the serializer guarantees many properties of the implementation that would otherwise have to be painfully extracted from a global analysis of the implementation.

The fundamental integrity constraint for mutual exclusion of the use of a resource is that if two requests $\text{SERIALIZER-REQUEST}_i$ and $\text{SERIALIZER-REQUEST}_j$ are made to the serializer then either the i-use completely precedes the j-use

$$\begin{tikzcd}
\text{RESOURCE-REQUEST}_i \arrow[d] & \\
\text{RESOURCE-REPLY}_i \arrow[r] & \text{RESOURCE-REQUEST}_j \arrow[d] \\
 & \text{RESOURCE-REPLY}_j
\end{tikzcd}$$

or the j-use completely precedes the i-use.

$$\begin{tikzcd}
 & \text{RESOURCE-REQUEST}_j \arrow[d] \\
\text{RESOURCE-REQUEST}_i \arrow[d] & \text{RESOURCE-REPLY}_j \arrow[l] \\
\text{RESOURCE-REPLY}_i &
\end{tikzcd}$$

which says that one process must enter and leave the protected resource before the other enters.

```
(mutual_exclusion_2 =resource) ≡                    ;to enforce mutual exclusion for a resource
  (create_serializer                                              ;construct a serializer
    (queues: q)                                                     ;with one queue q
    (crowds: c)                                                     ;and one crowd c
    (entry:                                 ;such that when entry is gained to the serializer
      (≡> =a_message                                                    ;with a message
        (enqueue q                                                      ;then wait on q
          (guarantee: (empty: c))                 ;for the condition that the crowd c is empty
          (then:                                                               ;then
            (relay_to resource                                         ;sent to the resource
              (message: a_message)               ;the message received by the serializer
              (thru: c)))))))                               ;passing through the crowd c
```

It is easy to see that mutual_exclusion_2 implements the integrity specification given above. The following invariant is true each time a process gains or releases possession:

$$((\text{size } c) \leq 1)$$

The only way to enter the resource through the serializer is to pass through the crowd c. Furthermore the crowd is guaranteed to be empty whenever a message is relayed to the resource.

Actually mutual_exclusion_2 implements a stronger specification: namely that if

$$\text{SERIALIZER-REQUEST}_i \dashrightarrow \text{SERIALIZER-REQUEST}_j$$

then

$$\begin{tikzcd}
\text{RESOURCE-REQUEST}_i \arrow[d] & \\
\text{RESOURCE-REPLY}_i \arrow[r, dashed] & \text{RESOURCES-REQUEST}_j
\end{tikzcd}$$

The proof of this stronger specification is given below. If we suppose that

$$\begin{tikzcd}
\text{SERIALIZER-REQUEST}_i \arrow[r] & \text{SERIALIZER-REQUEST}_j
\end{tikzcd}$$

then

$$\begin{tikzcd}
\text{SERIALIZER-REQUEST}_i \arrow[r] \arrow[d] & \text{SERIALIZER-REQUEST}_j \arrow[dd] \\
\text{ENTER}_i \arrow[d] & \\
\text{ENQUEUE}_i \arrow[r] & \text{ENTER}_j \arrow[d] \\
 & \text{ENQUEUE}_j
\end{tikzcd}$$

follows from mutual exclusion for serializers. Therefore

$$\begin{tikzcd}
\text{ENQUEUE}_i \arrow[d] & \\
\text{DEQUEUE}_i \arrow[r] & \text{ENQUEUE}_j \arrow[d] \\
 & \text{DEQUEUE}_j
\end{tikzcd}$$

since queues of a serializer are first-in-first-out. Since serializers are mutually exclusive we know that

$$\begin{tikzcd}
\text{DEQUEUE}_i \arrow[d] & \\
\text{JOIN-CROWD}_i \arrow[r] & \text{DEQUEUE}_j
\end{tikzcd}$$

Therefore

$$\begin{tikzcd}
\text{JOIN-CROWD}_i \arrow[d] & \\
\text{LEAVE-CROWD}_i \arrow[r] & \text{DEQUEUE}_j
\end{tikzcd}$$

since the prerequisite for DEQUEUE_j is that the crowd must be empty. Now we can read off the desired answer by transitivity:

$$\begin{tikzcd}
\text{JOIN-CROWD}_i \arrow[d] & \\
\text{RESOURCE-REQUEST}_i \arrow[d] & \\
\text{RESOURCE-REPLY}_i \arrow[d] & \\
\text{LEAVE-CROWD}_i \arrow[r] & \text{DEQUEUE}_j \arrow[d] \\
& \text{JOIN-CROWD}_j \arrow[d] \\
& \text{RESOURCE-REQUEST}_j
\end{tikzcd}$$

One extremely common specification is that an actor must reply to each request it receives (a guarantee of service that implies that the actor is starvation-free and thus free of deadlock). For serializers this is expressed in terms of events by simply requiring that for every **SERIALIZER-REQUEST** event there is a corresponding **SERIALIZER-REPLY** event in the history. Similarly the resource can be required to reply to requests by specifying that for every **RESOURCE-REQUEST** event there is a corresponding **RESOURCE-REPLY** event in the history. A serializer constructed using mutual_exclusion_2 is guaranteed to reply to requests provided that the resource it encases is guaranteed to reply to requests.

The code for mutual_exclusion_1 looks slightly shorter than the code for mutual_exclusion_2, but that is largely due to the extra words that indicate the structure of mutual_exclusion_2. In mutual_exclusion_1 the semaphore bookkeeping is explicit for the message passing to and from the resource. In mutual_exclusion_2 the required bookkeeping is implicit in the structure of the language construct.

SECTION VII --- READERS-WRITERS PROBLEMS

VII.1 --- Readers-Writers Integrity Specification

A readers-writers serializer is intended to protect the integrity of a resource by scheduling access to the resource in such a way that it is impossible for two processes to overlap in their use of the resource if one of them is a writer. An event in which a message is received by the protected resource is called a RESOURCE-REQUEST, of which there are two special cases: RESOURCE-READ and RESOURCE-WRITE. In response to these requests the protected resource will produce responses which will be called RESOURCE-REPLY events.

The integrity specification for a readers-writers serializer is that "a write request excludes all other requests from the resource". This integrity specification can be expressed in event terms as follows: if SERIALIZER-WRITE$_i$ and SERIALIZER-REQUEST$_j$ are two requests received by the serializer then either

RESOURCE-WRITE$_i$ --> RESOURCE-REPLY$_i$ -->
    RESOURCE-REQUEST$_j$ --> RESOURCE-REPLY$_j$

or

RESOURCE-REQUEST$_j$ --> RESOURCE-REPLY$_j$ -->
    RESOURCE-WRITE$_i$ --> RESOURCE-REPLY$_i$

VII.2 --- Readers-Writers Scheduling Specifications

Variations of the readers-writers problem derive from the desirability of imposing stronger scheduling specifications than simply that the serializer must reply to requests that it receives. Note that readers do not interfere with one another even if they are executing in parallel in the protected resource. Therefore allowing multiple readers into the resource concurrently can increase thruput. Several variations of scheduling specifications that require more concurrency will be presented below. In all of these implementations we will keep track of whether there are readers in the resource or there is a writer in the resource by keeping a separate crowd for the readers and a separate crowd for the writer (which will never have more than one member).

Below we will present several implementations of the readers-writers problems in which **readers** is the crowd of readers in the protected resource and **writer** is the crowd of writers in the protected resource. The following invariants are relevant to understanding all of the implementations to be presented below:

the size of the **writer** *crowd is never greater than one*
the **readers** *crowd and the* **writer** *crowd are never both nonempty at the same time*

The fact that all of the above invariants are preserved will be immediately evident from inspection of the code of all the implementations given below. No complicated chain of reasoning will be required. This is an example of how properties of a scheduler implemented using serializers can be seen with greater perspicuity than is possible from implementations using less structured arbitration primitives such as semaphores.

VII.3 --- Writer Excludes Others

Let us introduce no new constraints beyond the primary one of insisting that a writer has sole access to the resource. One possible implementation that would satisfy those constraints is the one-at-a-time serializer presented above. Another implementation follows that would allow readers to access the resource concurrently.

In the implementation below we provide that a resource which is to be scheduled for reading and writing will receive messages of the form (*read (using:* directions)) and (*write (using:* directions)). The directions included in these messages can be as complicated as desired up to and including an arbitrary procedure for carrying out the transaction on the protected resource without side effects. Note that this degree of generality in the directions can complicate verifying that the resource will reply to each request which it is sent. Nevertheless, in the discussion below we will assume that the resource will always reply to requests of the form (*read (using:* directions)) and (*write (using:* directions)).

```
((writer_exclude_others =the_resource) ≡      ;a serializer which enforces that writers excludes others
                                                   ;from the resource is implemented by constructing
 (create_serializer                                                          ;a serializer which has
   (queues: readers_q writers_q)                         ;two queues called readers_q and writers_q
   (crowds: readers writer)                                   ;two crowds called readers and writer
   (entry:                                                                          ;after entry
     (cases                                                    ;there are two cases for the message
      (≡> (read (using: =directions))          ;receive a request to read the resource using directions
          (enqueue readers_q                                     ;enqueue on the readers queue to
             (guarantee: (empty: writer))            ;guarantee that there is no writer in the resource
             (then:                                                                ;when dequeued,
                (relay_to the_resource                          ;deliver the message to the resource
                   (message: (read (using: directions)))
                   (thru: readers)))))                           ;passing thru the readers crowd
      (≡> (write (using: =directions))      ;receive a request to write in the resource using directions
          (enqueue writers_q                                         ;enqueue on the writers_q to
             (guarantee: (empty: readers writer))
                        ;guarantee that there are neither readers nor a writer in the resource
             (then:                                                                ;when dequeued,
                (relay_to the_resource                          ;deliver the message to the resource
                   (message: (write (using: directions)))
                   (thru: writer)))))))))                         ;passing thru the writer crowd
```

It is easy to see that the above implementation guarantees that writers will exclude others from the resource since if there is an element in the writer crowd then all the queues of the serializer are blocked.

```
                    ((size writer) ≤ 1)
          (if (nonempty: writer) then (empty: readers))
```

However, the above implementation does not satisfy the requirement that the serializer must always reply to requests which it receives since nothing forces the **readers** crowd to eventually become empty. If the readers crowd does not empty then a writer could be stuck forever in **writers_q** (this problem has its roots in the violation of our recommendation that no two processes should be "ready to go" in a serializer at the same time). Therefore the above implementation must be refined or interpreted in some way in order to meet the specifications.

VII.4 --- First Come First Served

The implementation given below satisfies the specification that the protected resource is served on a first come first served basis. In addition, starvation is not possible with the **first_come_first_served** serializer and a certain amount of concurrency is guaranteed. Note that the additional specification results in an implementation that is simpler than the previous one. The added simplicity is due to the ability of serializers to have processes waiting in a single queue for different conditions.

```
((first_come_first_served =the_resource) ≡        ;a first come first served serializer of the resource which
                                                                       ;can be implemented by constructing
   (create_serializer                                                          ;a serializer which has
      (queues: waiting_q)                                                   ;a queue called the waiting_q
      (crowds: readers writer)                                  ;and two crowds called readers and writer
      (entry:                                                                               ;after entry
       (cases                                                            ;there are two cases for the message
         (≡> (read (using: =directions))               ;receive a request to read the resource using directions
            (enqueue  waiting_q                                             ;enqueue on the waiting queue to
               (guarantee: (empty: writer))             ;guarantee that there is no writer in the resource
               (then:                                                                         ;when dequeued,
                  (relay_to the_resource                                 ;deliver the message to the resource
                     (message:  (read (using: directions)))
                     (thru: readers)))))                                   ;passing thru the readers crowd
         (≡> (write (using: =directions))        ;receive a request to write in the resource using directions
            (enqueue waiting_q                                              ;enqueue on the waiting queue to
               (guarantee: (empty: readers writer))
                              ;guarantee that there are neither readers nor a writer in the resource
               (then:                                                                         ;when dequeued,
                  (relay_to the_resource                                 ;deliver the message to the resource
                     (message: (write (using: directions)))
                     (thru: writer)))))))))                                          ;  thru the writer crowd
```

VII.4.a --- Requiring Concurrency in Implementations

In the readers-writers problems there is a basic integrity constraint that the serializer must maintain, which is to ensure that a writer does not have access to the resource at the same time as any other requestor. However, a simple one-at-a-time approach can easily guarantee this property. The more complex versions of the problem attempt to provide readers with concurrent access to the resource without starving the writers. When we say that some amount of concurrency is guaranteed, we mean that the specifications for the serializer require that certain readers be given the opportunity to access the resource at the same time.

Note that a serializer cannot guarantee that the requests to a protected resource are actually processed in parallel, since either the structure of the resource or some externally defined scheduling policy may prevent actual parallelism. We say that readers R_i and R_j are concurrent readers if

$\text{JOIN-CROWD}_i \text{ --> } \text{LEAVE-CROWD}_j$
and
$\text{JOIN-CROWD}_j \text{ --> } \text{LEAVE-CROWD}_i$

The specifications for the first-come-first-serve serializer include a requirement for concurrency. We can informally express this requirement as saying that whenever one reader's entry into the serializer (an ENTRY event) immediately precedes another reader's entry, and the second reader enters the serializer before the first reader enters the resource (a JOIN-CROWD event), then these two readers must concurrently be in the resource. We can also give a more formal specification in terms of events:

If R_i and R_j are readers such that
$\text{ENTRY}_i \text{ --> } \text{ENTRY}_j \text{ --> } \text{JOIN-CROWD}_i$
and
there is no requestor X_k (a reader or writer) such that
$\text{ENTRY}_i \text{ --> } \text{ENTRY}_k \text{ --> } \text{ENTRY}_j$

then R_i and R_j must be concurrent readers, i.e.

$\text{JOIN-CROWD}_i \text{ --> } \text{LEAVE-CROWD}_j$
and
$\text{JOIN-CROWD}_j \text{ --> } \text{LEAVE-CROWD}_i$

Note that the above requirement would be the same if we required that the requestor X_k be a writer, although the proof would be somewhat more difficult.

VII.4.b --- Proof of Guaranteed Concurrency

A proof that the **first_come_first_served** serializer shown above satisfies the given concurrency requirement proceeds by assuming the existence of two readers with the given relationship, then showing that they must be concurrent readers. Since we have

$$\text{ENTRY}_i \dashrightarrow \text{ENTRY}_j \dashrightarrow \text{JOIN-CROWD}_i$$

we know that the reader R_i must be in the **waiting_q** when the reader R_j gains possession of the serializer. R_j must be enqueued directly behind R_i, since by our assumptions there are no intervening entries to put other requestors in the **waiting_q**. Therefore when reader R_i does get into the resource through a JOIN-CROWD event (thereby releasing possession of the serializer), then the requestor at the head of the **waiting_q** must be R_j and the condition of (*empty:* **writers**) must be true. We then appeal to the priority which serializers give to processes waiting in interal queues of the serializer.

$$\text{JOIN-CROWD}_j \dashrightarrow \text{LEAVE-CROWD}_i$$

Since we know that JOIN-CROWD_i --> JOIN-CROWD_j, and JOIN-CROWD_j --> LEAVE-CROWD_j, we conclude that JOIN-CROWD_i --> LEAVE-CROWD_j, which completes the proof that R_i and R_j are concurrent readers.

VII.5 --- **Readers Priority**

The following serializer forces readers into the resource concurrently. However, we need to guard against starvation. Our approach is to allow all waiting readers to enter the resource, then to designate the writer which has been waiting as the new privileged writer, and keep further readers from entering the resource until the privileged writer has relayed its message to the resource. After the privileged writer has been served, then all readers which have been waiting for that writer to finish are allowed to enter the resource, and a new privileged writer is chosen. A reader may not deliver a message to the resource while there is a privileged writer, or there is a writer in the resource. A writer may not enter the resource unless it is a privileged writer, and there are neither readers nor a writer in the resource.

```
((read_concurrently =the_resource) ≡          ;a serializer which enforces concurrency among readers of
                                               ;the resource is implemented by constructing
 (create_serializer                            ;a serializer which has
   (queues: readers_q writers_q)               ;two queues called readers_q and writers_q
   (crowds: readers writer)                    ;two crowds called readers and writer
   (entry:                                     ;after entry
     (cases                                    ;there are two cases for the message
      (≡> (read (using: =directions))          ;receive a request to read the resource using directions
          (enqueue readers_q                   ;enqueue on the readers queue to
             (guarantee: (empty: writer))      ;guarantee that there are no writers in the resource
             (then:                            ;when dequeued,
                (relay_to the_resource         ;deliver the message to the resource
                   (message: (read (using: directions)))
                   (thru: readers)))))         ;passing thru the readers crowd
      (≡> (write (using: =directions))         ;receive a request to write in the resource using directions
          (enqueue writers_q                   ;enqueue on the writers queue to
             (guarantee: (empty: readers_q writer))
                                               ;guarantee that readers_q, and writer crowd are all empty
             (then:                            ;when dequeued,
                (enqueue readers_q             ;enqueue on readers_q to
                   (guarantee: (empty: readers writer))
                                ;guarantee that there are neither readers nor a writer in the resource
                   (then:                      ;when dequeued,
                      (relay_to the_resource   ;deliver the message to the resource
                         (message: (write (using: directions)))
                         (thru: writer)))))))))))   ;passing thru the writer crowd
```

The above implementation is a little more complicated than the previous one. However, it is not difficult to show that writers exclude others using the technique used for the previous implementation since the following invariants are maintained:

((size writer) ≤ 1)
(*if* (*nonempty:* writer) *then* (*empty:* readers))

In order to show that neither readers or writers can possible starve, consider the following "traffic diagram" for the queues and crowds of the serializer:

```
Entering readers               Entering writers
        |                             |
        |                             |
        |                             V
        |                         writers_q
        |                             |
        |                             |
        +----------+    +-------------+
                   |    |
                   |    | (empty: readers_q writer)
                   |    |
                   V    V
                  readers_q
                   |    |
  (empty: readers) |    | (empty: readers writer)
                   |    |
        +----------+    +-------------+
        |                             |
        |                             |
        |                             |
        |                             |
        V                             V
     readers                        writer
```

Traffic Diagram for Queues and Crowds of read_concurrently

The idea of the proof is to first show that the readers_q must eventually empty, then to show that any writer in the writers_q must eventually migrate to the readers_q. These two conditions ensure that every read or write request to the serializer is eventually satisfied.

VII.5.a --- Proof that the Readers Queue Must Empty

If there is a privileged writer in the readers_q, then there is only one such writer, and it must be at the head of the queue. A writer can only enter the readers_q after it has been dequeued from the writers_q, and the guarantee of every writer exiting the writers_q is that the readers_q is empty. Thus, not only is it true that there may be only one writer in the readers_q, the writer must also be at the head of the queue if it is there.

Processes may only enter the readers crowd or the writer crowd by first exiting the readers_q. We have assumed that the_resource is correct in that every message sent to the resource will eventually produce a single reply. Therefore, if a writer is at the head of the readers_q, it is guaranteed that no messages will be sent to **the_resource** from the serializer until both the **readers** crowd and the **writer** crowd are empty.

By a similar argument, if a reader is at the head of the **readers_q**, the **writer** crowd must eventually empty, which implies that a reader at the head of the **readers_q** must eventually exit the queue. Once a single reader is dequeued from the **readers_q** all readers in the **readers_q** in front of a writer must be dequeued. For any reader in the **readers_q** an event of the form $\text{ESTABLISHED}_{\text{readers\_q}}$ must be followed by an event of the form $\text{JOIN-CROWD}_{\text{reader}}$. For every such JOIN-CROWD event, the serializer is released, and if the **readers_q** has a reader at its head, that reader must be dequeued, since the **reader_q** is the only internal queue with its guaranteed condition true (the **writer** crowd remains empty).

Thus, we have shown that all readers must exit the **readers_q** if they occur before a writer in that queue or if there is no writer in that queue. Further, if a writer is at the head of the **readers_q** then it also must exit the **readers_q**. Once a writer exits the **readers_q** there may be no additional writers added to that queue until it is empty. Therefore, the **readers_q** must empty.

VII.5.b --- Proof That No Process in the writers_q Can Starve

The idea behind this proof is simple. Any writer in the **writers_q** is waiting for both the **writer** crowd and the **readers_q** must empty. By our assumption about the resource there must be a RESOURCE-REPLY message for any writer, so the **writer** crowd must empty. We have just proved above that the **readers_q** must empty. Therefore the process at the head of the **writers_q** must eventually be dequeued, which is sufficient to show that no process in the **writers_q** can starve.

VII.5.c --- Requiring Concurrency for Reader's Priority

The readers-priority serializer is intended to give more thruput to readers at the expense of writers, while still guaranteeing that each write request will receive a reply. Our informal requirement is whenever one reader's entry into the serializer follows another reader's entry (regardless of intervening serializer entries), and the second reader enters the serializer before the first reader enters the resource, then the two readers are concurrent within the resource. In terms of events, this requirement can be expressed as:

If R_i and R_j are readers such that

$$\text{ENTRY}_i \dashrightarrow \text{ENTRY}_j \dashrightarrow \text{JOIN-CROWD}_i$$

then R_i and R_j must be concurrent readers, i.e.

$$\text{JOIN-CROWD}_i \dashrightarrow \text{LEAVE-CROWD}_j$$

and

$$\text{JOIN-CROWD}_j \dashrightarrow \text{LEAVE-CROWD}_i$$

Note that the above requirement is stronger than the concurrency requirement given for first_come_first_served.

All readers that have entered the serializer and not yet in the resource are in the readers_q, and the only writer that can be in the readers_q must be at the head of the queue. We have previously shown that once one reader can leave the readers_q then all must leave the readers_q and enter the resource. We note that by the axiom of giving internal queues priority over external queues all readers in the readers_q must enter the resource before any reply from the resource will enter the serializer (through a LEAVE-CROWD event). This completes the proof.

SECTION VIII --- ANALYSIS of SERIALIZERS

VIII.1 --- Comparison with Monitors

Serializers are generalizations of the "secretary" concept which was conceived by Dijkstra and later developed into "monitors" by Brinch-Hansen, and Hoare. The purpose of serializers is to schedule access to shared resources in order to protect their integrity. Roughly speaking, serializers are analogous to monitors in the same way that actors are analogous to SIMULA-67 classes. A serializer is an actor that will allow only one process to have possession at a time whereas a monitor is a SIMULA-67 class that will allow only one process to be executing inside it at a time. A general principle of efficient operation that is applicable to both serializers and monitors is to try to keep the serializer [monitor] unlocked as much of the time as possible to keep it from being a bottleneck in the operation of a larger system.

We believe that serializers are easier to write and verify in a modular fashion than monitors. The correctness of a serializer is independent of the way in which it is used. It is more difficult to verify the correctness of a monitor since the ordering of monitor operations and access to the protected resource is dependent on the programs using the monitor. Serializers support modular programming better than monitors because serializers can be sensibly nested inside one another whereas usually it is unprofitable to nest one monitor inside another because the outer monitor will be tied up while the inner one is in use. A further advantage for serializers is that use of a protected resource appears identical to the use of the unprotected resource in a program that uses the resource.

Using serializers the condition associated with an *enqueue* command is explicitly stated in the code of the implementation whereas the conditions used in the WAIT statement in a monitor are syntactic constructs which do not explicitly state what condition must be guaranteed in order to proceed. A process which needs to wait for some condition to be guaranteed before it proceeds explicitly states the condition in the serializer. In our experience this feature lessens the number of internal queues and tends to simplify program proofs.

Another difference is the use of crowds rather than counters to keep track of processes that have been allowed to access the encapsulated resource. While there is an additional cost associated with such accounting, we believe that the benefits will make the use of crowds a decided advantage. It is possible to examine the crowds to determine which processes are currently accessing which resources. We expect that this will have significant impact on debugging programs that use parallel processes. Furthermore, if some process must be terminated externally, then the resources it is accessing can be reclaimed in a modular fashion.

Axioms about monitors and implementation of monitors are both made more difficult by the necessity of having an "urgent queue". Proofs of programs using the SIGNAL statement are made more complicated by the necessity of considering the possible effects of processes emerging from the urgent queue. Hoare states in his paper on monitors that he considers it to be good practice to exit a monitor immediately after a SIGNAL statement. We consider it to be a practice worthy of enforcement.

All internal queues in a serializer are explicit, and explicit signals do not occur. Rather, the conditions for emerging from a queue are explicitly listed at the point of enqueueing.

SECTION IX --- MORE GENERAL SERIALIZERS

Serializers need to be generalized in several ways in order to be a general purpose synchronization mechanism.

Thus far in this paper we have restricted our attention to serializers which use first-in first-out queues as the mechanism for scheduling the release of processes waiting for some condition. It is necessary to generalize this mechanism in order to conveniently solve certain scheduling problems. For example Hoare has used priority queues to solve the disk head scheduling problem. Below we present an implementation of a virtual disk which efficiently schedules disk accesses to a real physical disk. The operations on the virtual disk are exactly the same as the ones defined on the physical disk. The implementation given below uses an algorithm similar to one used by Hoare in an implementation using monitors. However, it will be possible to prove that our implementation using serializers always performs all disk operations which are requested. It is not possible to prove this property for the implementation using monitors because a process might request the disk and then never release it.

```
((virtual_disk =physical_disk) ≡        ;a virtual disk behaves like a physical disk with optimized head motion
   (let                                                                                              ;let
      (current_position initially 0)                                   ;the current position initialized to 0
      (current_direction initially up)                       ;the current direction of motion is initially up
    then
      (create_serializer                                                              ;create a serializer with
          (priority_queues: up_queue down_queue)                                         ;two priority queues
          (crowds: disk_users)                                                                    ;one crowd
          (entry:                                                                   ;after gaining possession
              (≡> (disk_request (operation: =op) (track: =track_num))
                                      ;receive disk request with specified operation and track number
                 (let                             ;define the procedures wait_up and wait_down as follows
                    ((wait_up) ≡                               ;the procedure wait_up is defined as follows
                       (enqueue up_queue (priority: track_number)
                                     ;enqueue in the up queue with priority given by the track number
                          (guarantee:                                                     ;to guarantee that
                             (and (empty: disk_users)                        ;no one is using the physical disk
                                (or (empty: down_queue)              ;and that either the down queue is empty
                                   (current_direction = up))))                  ;or an upsweep is in progress
                          (then: (current_position ← track_number)                  ;then update the position
                             (current_direction ← up)
                             (relay_to physical_disk                  ;relay the request to the physical disk
                                (message: (disk_request (operation: op) (track: track_num))
                                (thru: disk_users))))))                       ;passing thru the disk users crowd
                    ((wait_down) ≡                           ;the procedure wait_down is defined as follows
                       (enqueue down_queue                                       ;enqueue in the down queue
                          (priority: (number_of_tracks - track_number))
                          (guarantee:
                             (and (empty: disk_users) (or (empty: up_queue) (current_direction = down))))
                          (then: (current_position ← track_number)
                             (current-direction ← down)
                             (relay_to physical_disk
                                (message: (request (operation: op) (track: track_num)))
                                (thru: disk_users)))))
               then
                  (rules track_number                                                   ;if the track number is
                     (≡> (> current_position) (wait_up))
                                       ;greater than the current position then wait in the up queue
                     (≡> current_position       ;else if the track number is the same as the current position
                        (rules current_direction                                  ;then if the current direction
                           (≡> up (wait_down))                               ;is up then wait in the down queue
                           (≡> down (wait_up))))                                   ;else wait in the up queue
                     (≡> (< current_position) (wait_down)))))))))
                 ;else if the track number is less than the current position then wait in the down queue
```

The proof that the above implementation always performs disk operations which are requested is accomplished by showing that the disk head sweeps the disk in both the upward and the downward direction servicing all requests for the tracks which it passes over. There are two cases to the proof:

> Case 1: One of the queues is empty. This means that any requests in the other queue are served in order of smallest priority number. All of the requests will be served because any new entries in the nonempty queue must have higher priority numbers than the lowest priority number already in the queue. The priority numbers are bounded above by the number of disk tracks. Since the internal queues of a serializer have absolute priority for next possession of the serializer over external requests, it follows that only finitely many new requests can enter a queue during each physical disk operation. Therefore if one of the queues remains empty then the other will eventually empty.
>
> Case 2: Both the **up_queue** and the **down_queue** are nonempty. The queue to be served is determined by the **current_direction**. The argument used in Case 1 can be used to show that if **current_direction** is **up** then it will remain **up** until **up_queue** has emptied and **current_direction** will then be changed to **down**. An analogous result to the previous statement holds with "**up**" replaced by "**down**".

SECTION X --- FUTURE WORK

The simple specification that "starvation is prohibited" is common to almost all synchronization specifications. Yet serializers at present make no such guarantee. They do make it easier to prove that an implementation using a serializer is free of the danger of starvation by facilitating the proof that the serializer always replies to requests which it receives. We feel that research should continue to search for mechanisms that provide effective guarantees of such properties, yet also provide sufficient generality to cope with a wide range of problems.

Serializers have potential for use in robust systems in that more information is available for error recovery. The additional information is useful for implementing debugging features, deadlock detection, and gracefully backing processes out of protected resources.

SECTION XI --- CONCLUSIONS

In this paper we have introduced a modular arbitration primitive called a serializer which is a generalization and improvement of the monitor construct previously developed by Brinch-Hansen and Hoare. Serializers aid in the synthesis of modular synchronizers because their structure corresponds in a natural way to typical specifications for useful synchronizers. The structure imposed by using serializers provides important guarantees that aid in proving that the implementation meets its specifications.

The specifications for a serializer include integrity specifications relating the order of access to the type of access, and scheduling specifications to ensure that differing types of access occur in the proper order. Part of this ordering specification included a specification requiring that certain requestors must be given the opportunity to use the resource concurrently. In the readers-writers serializer, we gave the integrity specifications that readers and writers were mutually exclusive in accessing the resource, and that at most one writer could access the resource at a time. Our different solutions to the readers-writers problem resulted from different scheduling specifications.

We have attempted to explicitly introduce facilities into serializers that directly correspond to synchronization specifications. The constraint that the resource is not being accessed by either readers or writers when a writer enters is explicit in the code of our implementations, as is the requirement that no writers are accessing the resource when a reader enters. Serializers provide that the condition for a waiting process to proceed is explicitly stated. In this way integrity specifications can be directly expressed in the language. Scheduling specifications are more complicated. We have been able to use a specification language based on partial orders among events to good effect to express scheduling specifications. Furthermore, the structure of the serializers has enabled us to give simple intuitive proofs that various scheduling specifications are satisfied by implementations that use serializers.

SECTION XII --- ACKNOWLEDGEMENTS

This paper builds on the work of Irene Greif who developed behavioral specifications for all of the versions of the readers-writers problems given in this paper. Conversations with Ole-Johan Dahl, E. W. Dijkstra, Tony Hoare, James H. Morris, and John Reynolds have helped us to crystallize our ideas on these issues. Tony Hoare and Mark Laventhal made several useful suggestions that have materially improved the content of this paper. We would further like to thank the referees for their comments and suggestions.

The concept of a serializer which we have developed over the last year owes a tremendous intellectual debt to the monitors developed by Brinch-Hansen and Hoare. We have attempted to constructively build on their work to develop better structured arbitration primitive for actor systems. The development of serializers would not have been possible if monitors had not been previously developed. Brinch-Hansen has developed the programming language Concurrent Pascal which includes monitors. He has used the language in the design and implementation of an extremely modular single-user operating system called Solo. Dave Bustard and Tony Hoare at the Queen's University of Belfast have participated in the development of the programming language Parallel Programming Pascal which includes a mechanism called an envelope, which is related to our serializer mechanism.

A very preliminary version of this paper was presented at the ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication in March 1975. In December 1975 this paper was extensively revised to reflect improvements in serializers that were achieved in the PLASMA implementation meetings that fall in which Russ Atkinson, Mike Freiling, Carl Hewitt, Kenneth Kahn, Henry Lieberman, Marilyn McLennan, Ronald Pankiewicz, Brian Smith, and Aki Yonezawa were active participants. These meetings resulted in substantial improvements in our understanding of modular arbitration mechanisms. Building on the work of Irene Greif, Aki Yonezawa has developed improved notations for expressing contracts for the implementations in this paper.

SECTION XIII --- BIBLIOGRAPHY

Brinch Hansen, P. "The Programming Language Concurrent Pascal" IEEE Transactions on Software Engineering. June, 1975. pp 199-207.

Brinch Hansen, P. "The Solo Operating System" Software-Practice and Experience. April-June 1976. pp. 141-205.

Bustard, D. W. "Parallel Programming Pascal (PPP)" Version 1. Dept. of Computer Science. Queen's University of Belfast. November 1975.

Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes" Acta Informatica. 1971.

Goodman, N. "Coordination of Parallel Processes in the Actor Model of Computation". Laboratory of Computer Science TR 173. June, 1976.

Greif, I. "Semantics of Communicating Parallel Processes" MAC Technical Report TR-154. September 1975.

Greif, I. and Hewitt, C. "Actor Semantics of PLANNER-73" Proceedings of ACM SIGPLAN-SIGACT Conference. Palo Alto, California. January, 1975.

Hansen, P.B. "Operating System Principles" Prentice-Hall. 1973.

Hewitt, C., Bishop P., and Steiger, R. "A Universal Modular Actor Formalism for Artificial Intelligence" IJCAI-73. Stanford, Calif. Aug, 1973. pp. 235-245.

Hewitt, C. and Smith, B. "Towards a Programming Apprentice" IEEE Transactions on Software Engineering. SE-1, 1. March 1975. pp. 26-45.

Hewitt, C. "Protection and Synchronization in Actor Systems" ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication. March 24-25, 1975. Santa Monica, Calif.

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" CACM. October, 1975.

Hoare, C. A. R. "Language Hierarchies and Interfaces" Lecture Notes in Computer Science No. 46. Springer, 1976. pp 242-265.

Howard, J. H. "Proving Monitors" CACM. May, 1976. pp 273-278.